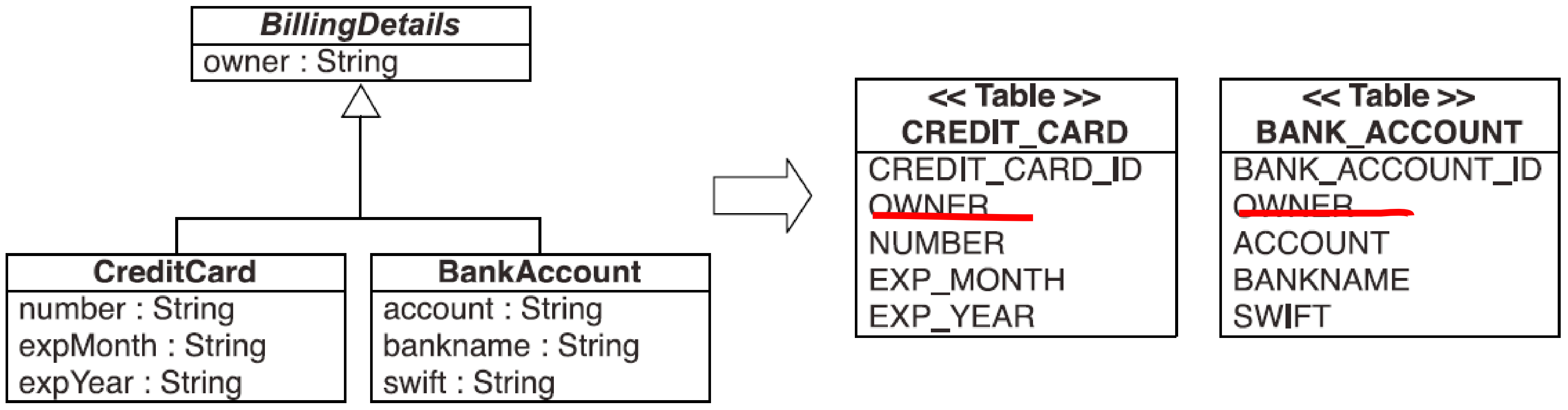
## Mapping class inheritance

- SQL database management systems don't support type inheritance—and even when it's available, it's usually proprietary or incomplete.
- There are four different approaches to representing an inheritance hierarchy:
  - Table per concrete class with implicit polymorphism—Use no explicit inheritance mapping, and default runtime polymorphic behavior.
  - Table per concrete class—Discard polymorphism and inheritance relationships completely from the SQL schema.
  - Table per class hierarchy—Enable polymorphism by denormalizing the SQL schema, and utilize a type discriminator column that holds type information.
  - Table per subclass—Represent is *a (inheritance) relationships as has a (foreign* key) relationships.

## 1- Table per concrete class with implicit polymorphism

All properties, including inherited properties, can be mapped to columns of this table



**Mapping all concrete classes to an independent table**

You don't have to do anything special in Hibernate to enable polymorphic behavior. The mapping for `CreditCard` and `BankAccount` is straightforward, each in its own entity `<class>` element, as we have done already for classes without a superclass (or persistent interfaces). Hibernate still knows about the superclass (or any interface) because it scans the persistent classes on startup.

- The main problem with this approach is that it doesn't support polymorphic associations very well.

- In the database, associations are usually represented as foreign key relationships.

- If the subclasses are all mapped to different tables, a polymorphic association to their superclass (abstract BillingDetails in this example) can't be represented as a simple foreign key relationship.

- This would be problematic in our domain model, because BillingDetails is associated with User; both subclass tables would need a foreign key reference to the USERS table.
- Or, if User had a many-to-one relationship with BillingDetails, the USERS table would need a single foreign key column, which would have to refer both concrete subclass tables.

- This isn't possible with regular foreign key constraints.

- Polymorphic queries (queries that return objects of all classes that match the interface of the queried class) are also problematic.

- A query against the superclass must be executed as several SQL SELECTs, one for each concrete subclass.

- For a query against the BillingDetails class Hibernate uses the following SQL:

  select CREDIT_CARD_ID, OWNER, NUMBER, EXP_MONTH, EXP_YEAR ... from CREDIT_CARD

  select BANK_ACCOUNT_ID, OWNER, ACCOUNT, BANKNAME, ...from BANK_ACCOUNT

- Notice that a separate query is needed for each concrete subclass.
- On the other hand, queries against the concrete classes are trivial and perform well—only one of the statements is needed.

- A further conceptual problem with this mapping strategy is that several different columns, of different tables, share exactly the same semantics.

- This makes schema evolution more complex.
- For example, a change to a superclass property results in changes to multiple columns.
- It also makes it much more difficult to implement database integrity constraints that apply to all subclasses.

- This approach is recommended (only) for the top level of your class hierarchy, where polymorphism isn't usually required, and when modification of the superclass in the future is unlikely.

## 2- Table per concrete class with unions

- First, let's consider a union subclass mapping with BillingDetails as an abstract class (or interface), as in the previous section.
- In this situation, we again have two tables and duplicate superclass columns in both:

  CREDIT_CARD and BANK_ACCOUNT.
- What's new is a special Hibernate mapping that includes the superclass, as you can see in the next slide.

**Using the `<union-subclass>` inheritance strategy**

```
<hibernate-mapping>
    <class
        name="BillingDetails"
        abstract="true">                  ❶

        <id                        ❷
            name="id"
            column="BILLING_DETAILS_ID"
            type="long">
            <generator class="native"/>
        </id>

        <property              ❸
            name="name"
            column="OWNER"
            type="string"/>

            ...

        <union-subclass            ❹
            name="CreditCard" table="CREDIT_CARD">

            <property name="number" column="NUMBER"/>
            <property name="expMonth" column="EXP_MONTH"/>
            <property name="expYear" column="EXP_YEAR"/>

        </union-subclass>

        <union-subclass
           name="BankAccount" table="BANK_ACCOUNT">
        ...

    </class>
</hibernate-mapping>
```

The first advantage you may notice with this strategy is the shared declaration of superclass (or interface) properties. No longer do you have to duplicate these mappings for all concrete classes—Hibernate takes care of this. Keep in mind that the SQL schema still isn't aware of the inheritance; effectively, we've mapped two unrelated tables to a more expressive class structure.

❶ An abstract superclass or an interface has to be declared as `abstract="true"`; otherwise a separate table for instances of the superclass is needed.

❷ The database identifier mapping is shared for all concrete classes in the hierarchy. The `CREDIT_CARD` and the `BANK_ACCOUNT` tables both have a `BILLING_DETAILS_ID` primary key column. The database identifier property now has to be shared for all subclasses; hence you have to move it into `BillingDetails` and remove it from `CreditCard` and `BankAccount`.

❸ Properties of the superclass (or interface) are declared here and inherited by all concrete class mappings. This avoids duplication of the same mapping.

❹ A concrete subclass is mapped to a table; the table inherits the superclass (or interface) identifier and other property mappings.

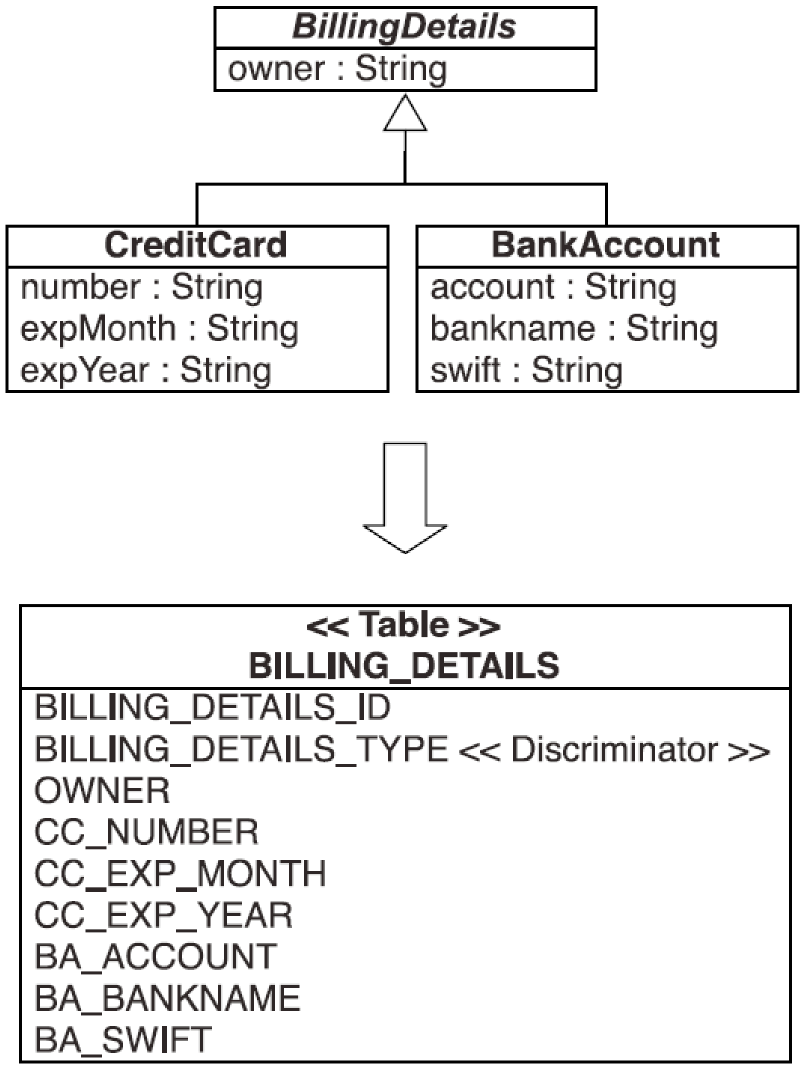
If your superclass is concrete, then an additional table is needed to hold instances of that class. We have to emphasize again that there is still no relationship between the database tables, except for the fact that they share some similar columns. The advantages of this mapping strategy are clearer if we examine polymorphic queries. For example, a query for `BillingDetails` executes the following SQL statement:

```
select
    BILLING_DETAILS_ID, OWNER,
    NUMBER, EXP_MONTH, EXP_YEAR,
    ACCOUNT, BANKNAME, SWIFT
    CLAZZ_
from
  ( select
      BILLING_DETAILS_ID, OWNER,
      NUMBER, EXP_MONTH, EXP_YEAR,
      null as ACCOUNT, null as BANKNAME, null as SWIFT,
      1 as CLAZZ_
    from
      CREDIT_CARD

    union

    select
        BILLING_DETAILS_ID, OWNER,
        null as NUMBER, null as EXP_MONTH, null as EXP_YEAR, ...
        ACCOUNT, BANKNAME, SWIFT,
        2 as CLAZZ_
      from
        BANK_ACCOUNT
   )
```

- Another much more important advantage is the ability to handle polymorphic associations.
- For example, an association mapping from User to BillingDetails would now be possible.
- Hibernate can use a UNION query to simulate a single table as the target of the association mapping.
- So far, the inheritance mapping strategies we’ve discussed don’t require extra consideration with regard to the SQL schema.
- No foreign keys are needed, and relations are properly normalized.

## 3- Table per class hierarchy

An entire class hierarchy can be mapped to a single table.



- There is one major problem: Columns for properties declared by subclasses must be declared to be nullable.
- If your subclasses each define several nonnullable properties, the loss of NOT NULL constraints may be a serious problem from the point of view of data integrity.
- Another important issue is normalization.
- As always, denormalization for performance can be misleading, because it sacrifices long-term stability, maintainability, and the integrity of data for immediate gains that may be also achieved by proper optimization of the SQL execution plans.

```
<hibernate-mapping>          ❶
    <class                  <┘
        name="BillingDetails"
        table="BILLING_DETAILS">

        <id
            name="id"
            column="BILLING_DETAILS_ID"
            type="long">
            <generator class="native"/>
        </id>
                                 ❷
        <discriminator         <┘
            column="BILLING_DETAILS_TYPE"
            type="string"/>
                                     ❸
        <property                  <┘
            name="owner"
            column="OWNER"
            type="string"/>

            ...
                            ❹
        <subclass         <┘
            name="CreditCard"
            discriminator-value="CC">

            <property name="number" column="CC_NUMBER"/>
            <property name="expMonth" column="CC_EXP_MONTH"/>
            <property name="expYear" column="CC_EXP_YEAR"/>

        </subclass>

        <subclass
            name="BankAccount"
            discriminator-value="BA">
        ...

    </class>
</hibernate-mapping>
```

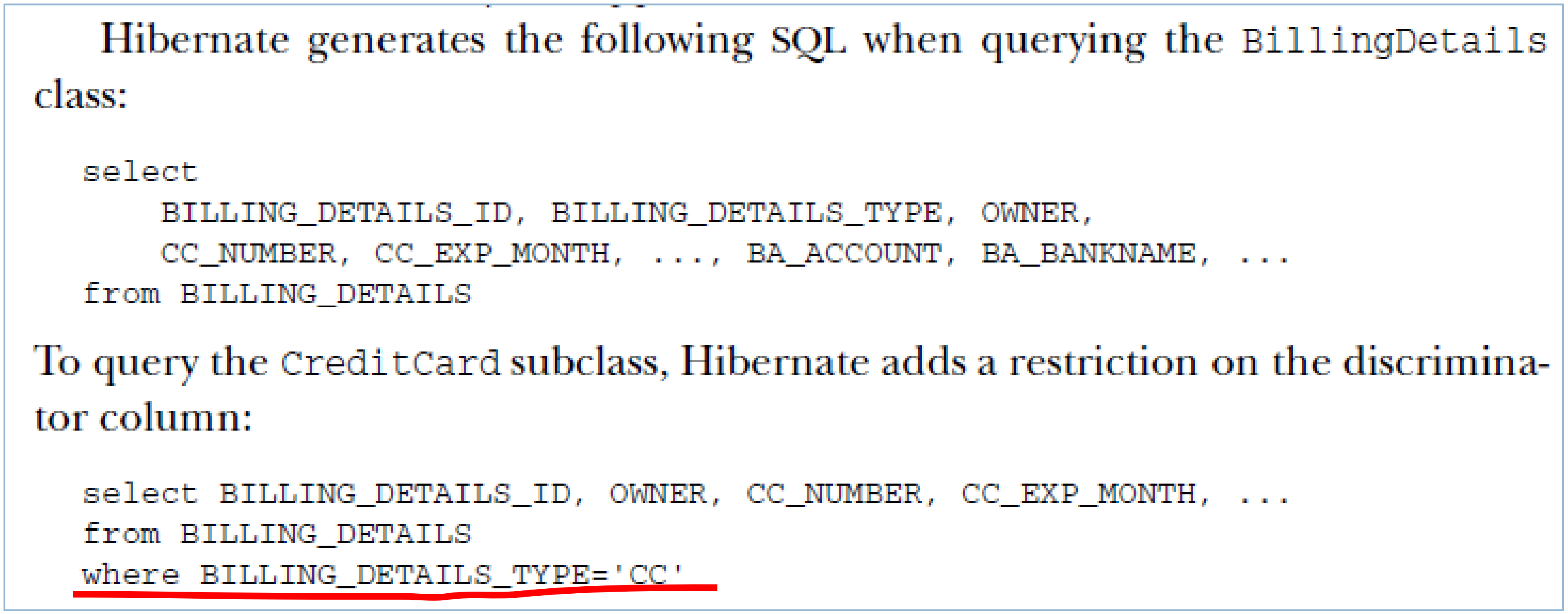
❶ The root class `BillingDetails` of the inheritance hierarchy is mapped to the table `BILLING_DETAILS`.

❷ You have to add a special column to distinguish between persistent classes: the discriminator. This isn't a property of the persistent class; it's used internally by Hibernate. The column name is `BILLING_DETAILS_TYPE`, and the values are strings—in this case, "CC" or "BA". Hibernate automatically sets and retrieves the discriminator values.

❸ Properties of the superclass are mapped as always, with a simple `<property>` element.

❹ Every subclass has its own `<subclass>` element. Properties of a subclass are mapped to columns in the `BILLING_DETAILS` table. Remember that `NOT NULL` constraints aren't allowed, because a `BankAccount` instance won't have an `expMonth` property, and the `CC_EXP_MONTH` field must be `NULL` for that row.

The `<subclass>` element can in turn contain other nested `<subclass>` elements, until the whole hierarchy is mapped to the table.



Hibernate generates the following SQL when querying the `BillingDetails` class:

```
select
    BILLING_DETAILS_ID, BILLING_DETAILS_TYPE, OWNER,
    CC_NUMBER, CC_EXP_MONTH, ..., BA_ACCOUNT, BA_BANKNAME, ...
from BILLING_DETAILS
```

To query the `CreditCard` subclass, Hibernate adds a restriction on the discriminator column:

```
select BILLING_DETAILS_ID, OWNER, CC_NUMBER, CC_EXP_MONTH, ...
from BILLING_DETAILS
where BILLING_DETAILS_TYPE='CC'
```

- The disadvantages of the table per class hierarchy strategy may be too serious for your design—after all, denormalized schemas can become a major burden in the long run.

- Your DBA may not like it at all.

- The next inheritance mapping strategy doesn’t expose you to this problem.

## 4- Table per subclass

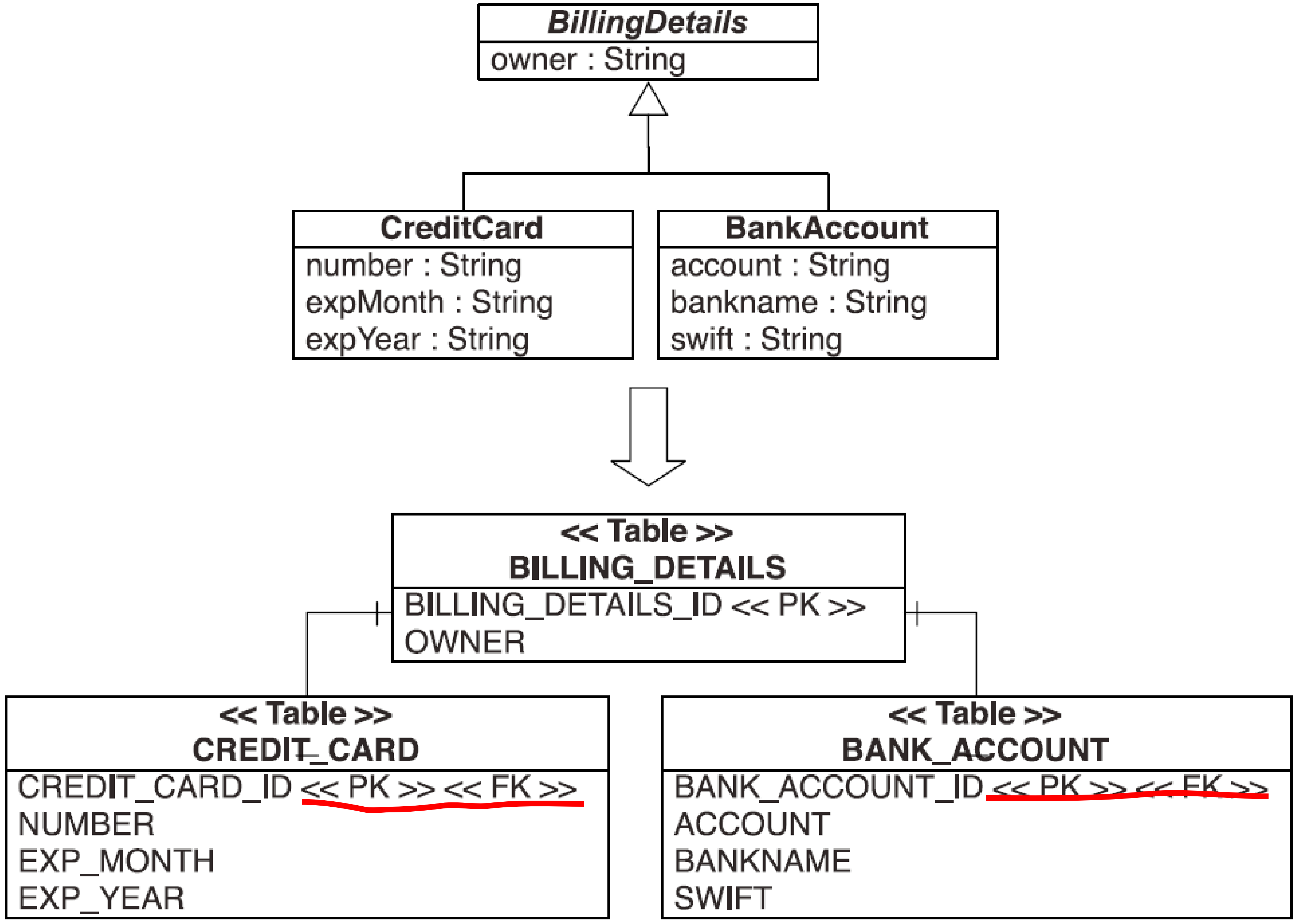
- The fourth option is to represent inheritance relationships as relational foreign key associations.
- Every class/subclass that declares persistent has its own table.
- Unlike the table per concrete class strategy we mapped first, the table here contains columns only for each noninherited property (each property declared by the subclass itself) along with a primary key that is also a foreign key of the superclass table.

- If an instance of the CreditCard subclass is made persistent,
  - the values of properties declared by the BillingDetails superclass are persisted to a new row of the BILLING_DETAILS table.
  - Only the values of properties declared by the subclass are persisted to a new row of the CREDIT_CARD table.

- The two rows are linked together by their shared primary key value.

- Later, the subclass instance may be retrieved from the database by joining the subclass table with the superclass table.

- The primary advantage of this strategy:
  - the SQL schema is normalized.



**Mapping all classes of the hierarchy to their own table**

```
<hibernate-mapping>
    <class                  ❶
        name="BillingDetails"
        table="BILLING_DETAILS">

        <id
            name="id"
            column="BILLING_DETAILS_ID"
            type="long">
            <generator class="native"/>
        </id>

        <property
            name="owner"
            column="OWNER"
            type="string"/>

            ...

        <joined-subclass        ❷
            name="CreditCard"
            table="CREDIT_CARD">

            <key column="CREDIT_CARD_ID"/>      ❸

            <property name="number" column="NUMBER"/>
            <property name="expMonth" column="EXP_MONTH"/>
            <property name="expYear" column="EXP_YEAR"/>

        </joined-subclass>

        <joined-subclass
            name="BankAccount"
            table="BANK_ACCOUNT">
        ...

    </class>
</hibernate-mapping>
```

❶ The root class `BillingDetails` is mapped to the table `BILLING_DETAILS`. Note that no discriminator is required with this strategy.

❷ The new `<joined-subclass>` element maps a subclass to a new table—in this example, `CREDIT_CARD`. All properties declared in the joined subclass are mapped to this table.

❸ A primary key is required for the `CREDIT_CARD` table. This column also has a foreign key constraint to the primary key of the `BILLING_DETAILS` table. A `CreditCard` object lookup requires a join of both tables. A `<joined-subclass>` element may contain other nested `<joined-subclass>` elements, until the whole hierarchy has been mapped.

Hibernate relies on an outer join when querying the `BillingDetails` class:

```
select BD.BILLING_DETAILS_ID, BD.OWNER,
       CC.NUMBER, CC.EXP_MONTH, ..., BA.ACCOUNT, BA.BANKNAME, ...
    case
        when CC.CREDIT_CARD_ID is not null then 1
        when BA.BANK_ACCOUNT_ID is not null then 2
        when BD.BILLING_DETAILS_ID is not null then 0
    end as CLAZZ_
from BILLING_DETAILS BD
    left join CREDIT_CARD CC
      on BD.BILLING_DETAILS_ID = CC.CREDIT_CARD_ID
    left join BANK_ACCOUNT BA
      on BD.BILLING_DETAILS_ID = BA.BANK_ACCOUNT_ID
```

The SQL `CASE` statement detects the existence (or absence) of rows in the subclass tables `CREDIT_CARD` and `BANK_ACCOUNT`, so Hibernate can determine the concrete subclass for a particular row of the `BILLING_DETAILS` table.

To narrow the query to the subclass, Hibernate uses an inner join:

```
select BD.BILLING_DETAILS_ID, BD.OWNER, CC.NUMBER, ...
from CREDIT_CARD CC
    inner join BILLING_DETAILS BD
      on BD.BILLING_DETAILS_ID = CC.CREDIT_CARD_ID
```

As you can see, this mapping strategy is more difficult to implement by hand—even ad-hoc reporting is more complex. This is an important consideration if you plan to mix Hibernate code with handwritten SQL.

Furthermore, even though this mapping strategy is deceptively simple, our experience is that performance can be unacceptable for complex class hierarchies.

## Mixing inheritance strategies

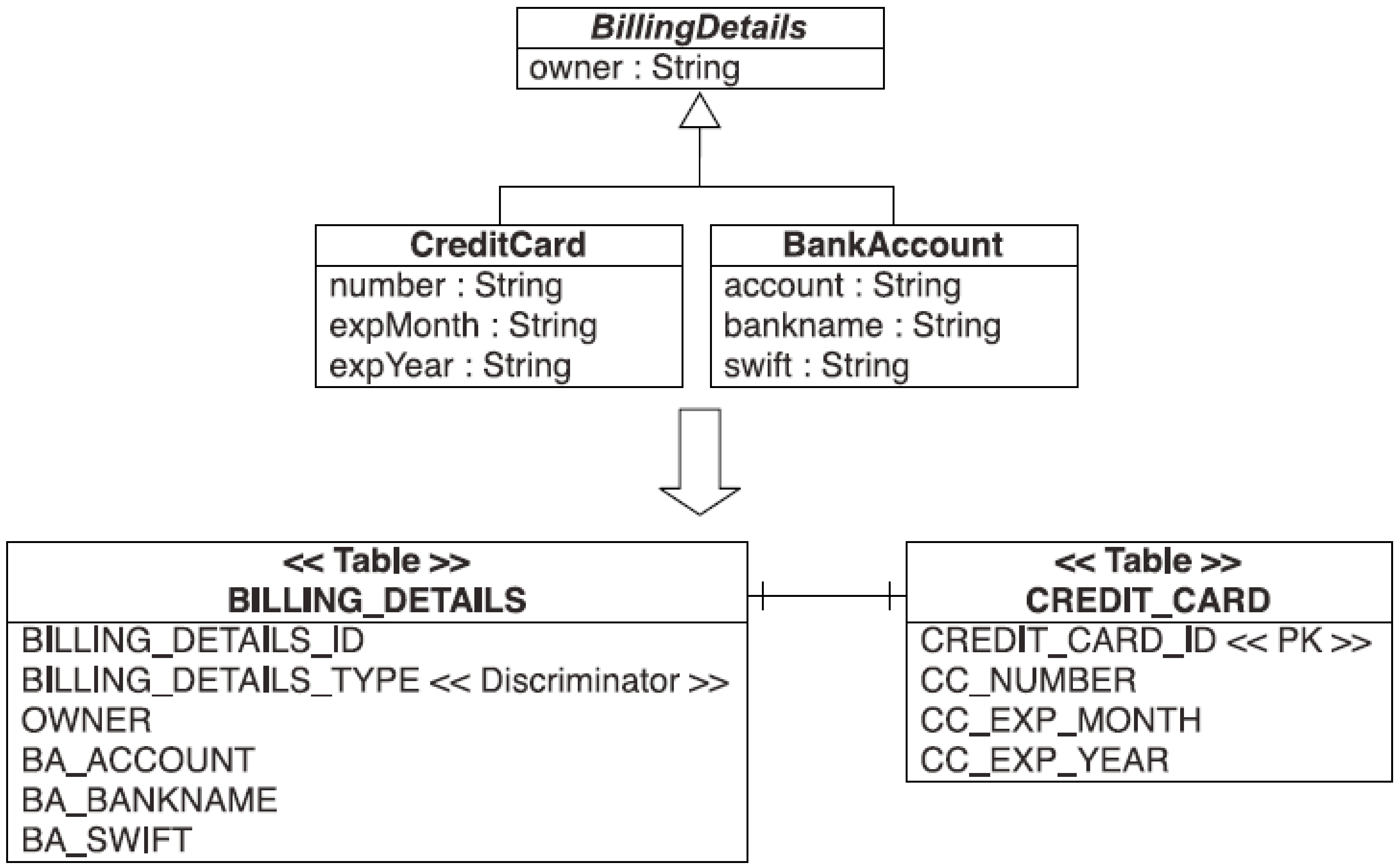
- You can map whole inheritance hierarchies by nesting <union-subclass>, <subclass>, and <joined-subclass> mapping elements.
- You can't mix them
  - for example, to switch from a table-per-class hierarchy with a discriminator to a normalized table-per-subclass strategy.
- Once you've made a decision for an inheritance strategy, you have to stick to it.
- This isn't completely true, however. With some Hibernate tricks, you can switch the mapping strategy for a particular subclass.
  - For example, you can map a class hierarchy to a single table, but for a particular subclass, switch to a separate table with a foreign key mapping strategy, just as with table per subclass.
- This is possible with the <join> mapping element:

```
<hibernate-mapping>
<class name="BillingDetails"
      table="BILLING_DETAILS">

    <id>...</id>

    <discriminator
        column="BILLING_DETAILS_TYPE"
        type="string"/>
        ...

    <subclass
        name="CreditCard"
        discriminator-value="CC">

        <join table="CREDIT_CARD">
            <key column="CREDIT_CARD_ID"/>

            <property name="number" column="CC_NUMBER"/>
            <property name="expMonth" column="CC_EXP_MONTH"/>
            <property name="expYear" column="CC_EXP_YEAR"/>
            ...
        </join>

    </subclass>

    <subclass
        name="BankAccount"
        discriminator-value="BA">

        <property name=account" column="BA_ACCOUNT"/>
        ...
    </subclass>

...

</class>
</hibernate-mapping>
```

- The <join> element groups some properties and tells Hibernate to get them from a secondary table.
- This mapping element has many uses.
- In this example, it separates the CreditCard properties from the table per hierarchy into the CREDIT_CARD table.
- The CREDIT_CARD_ID column of this table is at the same time the primary key, and it has a foreign key constraint referencing the BILLING_DETAILS_ID of the hierarchy table.
- The BankAccount subclass is mapped to the hierarchy table.

At runtime, Hibernate executes an outer join to fetch `BillingDetails` and all subclass instances polymorphically:

```
select
    BILLING_DETAILS_ID, BILLING_DETAILS_TYPE, OWNER,
    CC.CC_NUMBER, CC.CC_EXP_MONTH, CC.CC_EXP_YEAR,
    BA_ACCOUNT, BA_BANKNAME, BA_SWIFT

from
    BILLING_DETAILS
left outer join
    CREDIT_CARD CC
        on BILLING_DETAILS_ID = CC.CREDIT_CARD_ID
```



**Breaking out a subclass to its own secondary table**

You can also use the `<join>` trick for other subclasses in your class hierarchy. However, if you have an exceptionally wide class hierarchy, the outer join can become a problem. Some database systems (Oracle, for example) limit the number of tables in an outer join operation. For a wide hierarchy, you may want to switch to a different fetching strategy that executes an immediate second select instead of an outer join:

```
<subclass
    name="CreditCard"
    discriminator-value="CC">

    <join table="CREDIT_CARD" fetch="select">
        <key column="CREDIT_CARD_ID"/>
        ...
    </join>

</subclass>
```

## Choosing a strategy

- You can apply all mapping strategies to abstract classes and interfaces.
- Interfaces may have no state but may contain accessor method declarations, so they can be treated like abstract classes.
- You can map an interface with <class>, <union-subclass>, <subclass>, or <joined-subclass>, and you can map any declared or inherited property with <property>.
- Hibernate won’t try to instantiate an abstract class, even if you query or load it.

## Rules of Thumb

- If you don't require polymorphic associations or queries,
  - lean toward tableper-concrete-class—in other words, if you never or rarely query for BillingDetails and you have no class that has an association to BillingDetails (our model has).
- If you do require polymorphic associations (an association to a superclass, hence to all classes in the hierarchy with dynamic resolution of the concrete class at runtime) or queries, and subclasses declare relatively few properties (particularly if the main difference between subclasses is in their behavior),
  - lean toward table-per-class-hierarchy.
  - Your goal is to minimize the number of nullable columns and to convince yourself that a denormalized schema won't create problems in the long run.
- If you do require polymorphic associations or queries, and subclasses declare many properties (subclasses differ mainly by the data they hold),
  - lean toward table-per-subclass.
  - Or, depending on the width and depth of your inheritance hierarchy and the possible cost of joins versus unions, use table-per-concrete-class.

- By default, choose table-per-class-hierarchy only for simple problems.
- For more complex cases, you should consider the table-per-subclass strategy.
- Complex inheritance is often best avoided for all sorts of reasons unrelated to ORM.

- Hibernate acts as a buffer between the domain and relational models, but that doesn't mean you can ignore persistence concerns when designing your classes.

- When you start thinking about mixing inheritance strategies, remember that implicit polymorphism in Hibernate is smart enough to handle more exotic cases.
  - For example, consider an additional interface in our application, Electronic-PaymentOption. This is a business interface that doesn't have a persistence aspect—except that in our application, a persistent class such as CreditCard will likely implement this interface. No matter how you map the BillingDetails hierarchy, Hibernate can answer a query from ElectronicPaymentOption correctly. This even works if other classes, which aren't part of the BillingDetails hierarchy, are mapped persistent and implement this interface.

- Hibernate always know what tables to query, which instances to construct, and how to return a polymorphic result.

- Finally, you can also use <union-subclass>, <subclass>, and <joined-subclass> mapping elements in a separate mapping file (as a top-level element instead of <class>).

- You then have to declare the class that is extended, such as
  <subclass name="CreditCard" extends="BillingDetails">,
  and the superclass mapping must be loaded programmatically before the subclass mapping file (you don’t have to worry about this order when you list mapping resources in the XML configuration file).

- This technique allows you to extend a class hierarchy without modifying the mapping file of the superclass.